REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1996	3. REPORT TYPE AND DATES COVERED technical

4. TITLE AND SUBTITLE IPT: An Object Oriented Toolkit for Interprocess Communication	5. FUNDING NUMBERS DAAE07-90-C-R059 TEC: DACA76-89-C0014
6. AUTHOR(S) Jay Gowdy	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The Robotics Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU-RI-TR-96-07
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) TACOM and Topographic Engineering Center	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; Distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

no abstract

14. SUBJECT TERMS	15. NUMBER OF PAGES 47 pp
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT unlimited	18. SECURITY CLASSIFICATION OF THIS PAGE unlimited	19. SECURITY CLASSIFICATION OF ABSTRACT unlimited	20. LIMITATION OF ABSTRACT unlimited

DTIC QUALITY INSPECTED 4

IPT: An Object Oriented Toolkit for Interprocess Communication

Jay Gowdy

CMU-RI-TR-96-07

THE ROBOTICS INSTITUTE

Carnegie Mellon University

The Robotics Institute

Technical Report

19960719 004

IPT: An Object Oriented Toolkit for Interprocess Communication

Jay Gowdy

CMU-RI-TR-96-07

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

March 1996

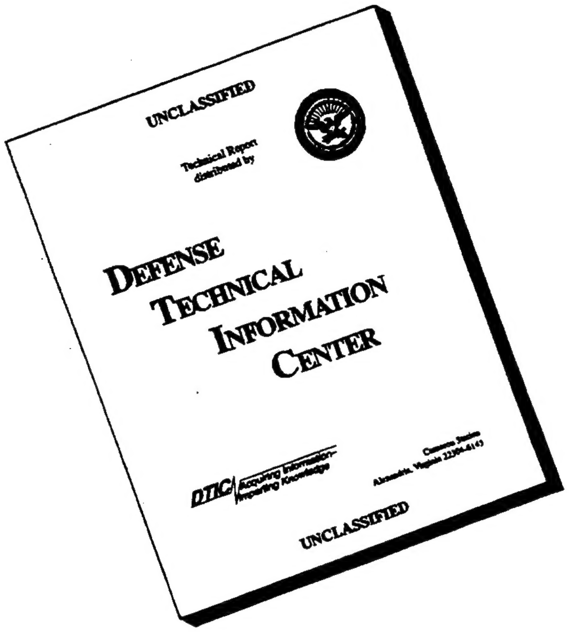
© 1995 Carnegie Mellon University

This work is funded under TACOM contract DAAE07-90-C-R059 and Topographic Engineering Center contract DACA76-89-C0014.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

DTIC QUALITY INSPECTED 4

DISCLAIMER NOTICE


UNCLASSIFIED
Technical Report distributed by
DEFENSE TECHNICAL INFORMATION CENTER
DTIC Acquiring Information-Imparting Knowledge
UNCLASSIFIED

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Table of contents

Introduction

IPT is an object oriented InterProcess communications Toolkit which has its roots in the TCX communications toolkit. As with TCX, IPT uses a message based paradigm to connect various processes, or modules, in a system together. Modules use IPT to establish connections between themselves and to send and receive messages. These connections can be considered as direct lines of communications setting up point to point links between modules without going through any "center."

Just as in TCX, each message has an instance number and a type. The instance number can be used to keep track of the sequencing of messages, and the type contains information about how the message is formatted. Message data can be formatted to allow unpacking into C or C++ structures. Messages can be handled by user defined handling routines or by searching through a message queue.

Both IPT and TCX have the same idea of a central communications process. These central processes are not the means by which all modules communicate, but they are the means by which all modules initiate communications. The IPT server has three jobs

1. To establish a consistent mapping between message type names (which are strings) and message type ID's (which are integers). This mapping means that each message will have associated with it a four byte integer rather than an arbitrarily long type name. Having the server make this mapping ensures consistency across all modules that the server works with.
2. To act as an telephone operator connecting modules together. A module can thus request a connection to another module by name without having to know what machine that other module is on or how to connect to that other module.
3. To act as a central repository for log information. For example, individual IPT modules can be run in "logging mode." In logging mode an IPT module will send the headers of all messages it sends and receives to the IPT server. A system developer can use this log information to track down problems in the system.

Once modules are connected, the server doesn't take up any more CPU cycles. It doesn't die, because IPT is a dynamic system. Modules are allowed to connect and disconnect throughout the lifetime of the system, and the server needs to be around in order to make and break these connections in an orderly fashion.

How is IPT different than TCX? For the most part, the functionality of TCX 8.5 (the most recent TCX used in the UGV program) can be considered a subset of the functionality of IPT. In fact, there are a set of C cover functions implemented in IPT that mimic TCX to make conversion from TCX to IPT easier. Here are some things that IPT can do that TCX cannot

1. Pigeon-hole messages. In TCX, if you wanted the most recent message of a certain type from a certain connection, you needed to do all sorts of processing. IPT provides a way to declare a message type a pigeon holed message type. Declaring a message type a pigeon hole means that if a handler is called or the message queues are searched for a message of this type you are guaranteed to get the most recent message of that type from a particular connection without any additional user processing.

2. Better access to the raw data. In both TCX and IPT, messages really consist of a length **n**, and then **n** bytes of data. TCX was not designed to give out the data in this form, event though some users want it in this form without going through the data formatting process. IPT supports both modes of interpreting messages equally, whereas in TCX the raw data access was a dangerous afterthought.
3. Multiple IPT servers. With IPT, you can set up several IPT domains, each with its own server. These domains can be linked together to allow clients to make inter-domain connections.
4. Built in support for client/server relationships. IPT gives you mechanisms to build and maintain client/server relationships among your modules. It will take care of the bookkeeping needed to maintain the client/server relationships when the clients or the servers die and restart.
5. Built in support for publisher/subscriber relationships. IPT gives you a mechanism whereby a module can declare that it is a publisher for a type. Subscriber modules can connect to this module and subscribe to the type. Then, when the publisher wants to update the data, it can "publish" it to all the modules that have registered as needing it. IPT also makes sure these publisher/subscriber relationships last when publishing and subscribing modules die.
6. Vastly more efficient communications (especially within a machine). IPT uses UNIX sockets instead of TCP/IP sockets for communications between modules running on the same UNIX machine, and IPT reduces the amount of data allocation and copying to a minimum. IPT has been clocked at 5 times faster than TCX for large messages on a single machine.

IPT also has some developmental advantages over TCX 8.5. IPT is written in C++ while TCX is written C. We provide C cover functions to allow C programmers to use the full power of IPT, but it is still written in an object oriented fashion in a more or less object oriented language. The advantage is in developmental flexibility. For example, many of the connections between IPT modules are instances of the class **TCPConnection**. A **TCPConnection** uses TCP/IP socket based communications to send and receive messages primitives. Now, a **TCPConnection** is a sub-class of **IPConnection**. We can implement shared-memory connections on real-time machines which will be vastly more efficient than TCP connections by creating another sub-class of **IPConnection** which implements the primitives in a different way. **TCPConnection**s and **SharedMemConnections** can coexist in the same process by taking advantage of some of the basic properties of C++. This kind of extension of TCX was virtually impossible due to the implementation language and developmental approach. This kind of extension of IPT falls naturally out the way it is designed and implemented.

Configuring IPT

The IPT source, libraries, binaries, and header files should be located in some central area.IPT uses the environment variable **INSTALL_DIR** to define where to install and access IPT include files, binaries, and libraries. For example, at CMU **INSTALL_DIR** is **/usr/projects/hmmwv**, so the library **$(INSTALL_DIR)/lib** would really be **/usr/projects/hmmwv/lib**.

IPT v6.4 comes with a "configure" script. This script lives in **$(INSTALL_DIR)/src/communications/ipt/configure** and allows you to specify what architectures and compilers you want to build IPT for and sets up subdirectories and links as needed.

The idea behind the configure script is that you want to be able to keep one copy of the source code while generating libraries and executables for many different platforms, compilers, and architectures. Each different platform, compiler, or architecture will have a "type name," for example, the type name associated with Cfront C++ translating compilers on UNIX is **cfront**. This means all object files, executables, and libraries are built in **$(INSTALL_DIR)/src/communications/ipt/cfront**. The configure script will automatically create this directory (if it does not already exist). The configure script will also automatically create the directory **$(INSTALL_DIR)/lib/cfront** so that the Cfront library will have a place to be installed. The configure script will also create a symbolic link **$(INSTALL_DIR)/lib/libcfrontipt.a** which points to **$(INSTALL_DIR)/lib/cfront/libipt.a**.

Each type must have a definition file that sets up environment variables and defines rules that are needed to compile to that type. The type file must be *config/<type name>.def*. For example, for the Cfront type's definition file is in **config/cfront.def.**

The configure script also lets you choose a "main" type. The IPT server built by the main type will be the only IPT server placed in **$(INSTALL_DIR)/bin**. A symbolic link **$(INSTALL_DIR)/lib/libipt.a** will be created which points to the main type's IPT library.

The configure script takes the following options:

-unix *<list of UNIX types>*

Defaults to "**cfront gcc**"

-vx *<list of VxWorks types>*

Defaults to "**vx**"

-install *<list of types to install>*

Defaults to the concatenation of all the UNIX and VxWorks types

-main *<the main type>*

Defaults to **cfront**

So, for example, running configure with no arguments is equivalent to

```
./configure -unix "cfront gcc" -vx "vx" -install "cfront gcc vx"
         -main "cfront"
```

This says that we will build a Makefile that can be used to build a Cfront based IPT, a Gnu based IPT, and a VxWorks IPT all on the same architecture. All three types will be made with a "**make install**," but the primary type is the **cfront** type. The Cfront type will be the one used to build the IPT server, and if anyone just links with **$(INSTALL_DIR)/lib/libipt.a**, it is the type that they will get.

As another example, I will give the command line that I use in configuring IPT. I need to primarily support Cfront, GNU, and Sparc based VxWorks types, but I also need to support compilation on Silicon Graphic's sharing the same file system. The Silicon Graphics code cannot be built during a single install phase, since I am installing the other types for use with Sun workstations. So, the configure line I use is

```
./configure -unix "cfront gcc SGI" -install "cfront gcc vx"
```

This means when I do a "**make install**" on a Sun, I build the Cfront, Gnu based, and Sparc VxWorks types. To compile the Silicon Graphics type I telnet to a Silicon Graphics machine and type

```
make install.SGI
```

As one more example, say you do not want to install any VxWorks types. Then you would simply do

```
./configure -vx ""
```

The UNIX types would still default to "**cfront gcc**", so you would only build the Cfront and Gnu types.

Similarly, if you wanted to add a version that used 68K cross compilers, but still shared the file system you would have to first define the file **config/vx68k.def** (you could probably use **config/vx.def** as a template), and then configure with,

```
./configure -vx "vx vx68k"
```

Compiling with IPT

The header files for IPT are in **$(INSTALL_DIR)/include/ipt**. The standard method is to pass
-I$(INSTALL_DIR)/include
to your compiler and then include **<ipt/*filename*.h>**.

If you are programming in C or C++, then you should include **<ipt/ipt.h>** in your files that use IPT. If you are programming in C++ then the files that you include vary depending on how much of IPT you use. You will always need **<ipt/ipt.h>**, and you will usually need **<ipt/message.h>** if you do any message processing.

IPT is written in C++, which presents a problem. There are two common classes of C++ compilers that we support for IPT. First, there is the Gnu family of compilers and second there is the Cfront family of compilers. The two classes of compilers differ in how they allocate and deallocate memory and in how they "mangle" C++ names, so it would be difficult and unpredictable to compile a program with g++ or gcc if the IPT library had been compiled with CenterLine C++ or the Sun Unbundled C++ compiler. Similarly, it would be impossible to compile a program with cc or a Cfront C++ translator if the IPT library had been compiled with g++.

To get around this we provide two IPT libraries. The Gnu library is in **$(INSTALL_DIR)/lib/gcc/libipt.a** and the Cfront library is in **$(INSTALL_DIR)/lib/cfront/libipt.a**. For convenience the configure script builds some symbolic links.

Now, if you are compiling the final stage with **cc**, you need to pass in to the linker either

$(INSTALL_DIR)/lib/cfront/libipt.a -lC

or (using the symbolic links set up by the configure script),

-L$(INSTALL_DIR)/lib -lipt -lC

The **-lC** is needed to tell the C linker that it needs to look in some of the C++ libraries. The only code needed from the C++ libraries is the code needed to implement the memory allocation and deallocation. If the C++ library, **libC.a**, is not installed in a standard directory (such as **/usr/local/lib**), you may have to either edit your **LD_LIBRARY_PATH** environment variable or place a **-L$(*CC_LIB_HOME*)** in your Makefile (where ***CC_LIB_HOME*** is the directory where **libC.a** exists).

If you are compiling the final stage with some form of CC (i.e., a cfront C++ translator), then you need to pass in either

$(INSTALL_DIR)/lib/cfront/libipt.a

or,

-L$(INSTALL_DIR)/lib -lipt

If you are using gcc or g++ then you need to pass in either

$(INSTALL_DIR)/lib/gcc/libipt.a

or,

-L$(INSTALL_DIR)/lib -lgccipt

Note that with gcc you don't need **-lC** because the memory allocation/deallocation routines for C++ are built into the libraries used by both gcc and g++.

Converting from TCX to IPT

There are a set of header functions in IPT that duplicate the functionality of TCX, right down to some of the dangerously unpredictable things it does with the raw data level. Any code that does not need the TCX private header file **tcxP.h** to compile will work the first time with IPT. Any code that does need **tcxP.h** will need more extensive work.

While you could just continue to use the original TCX header file and just re-link we suggest using the IPT TCX cover header file. To do this transparently set up a link in **$(INSTALL_DIR)/include/tcx.h** to **$(INSTALL_DIR)/include/ipt/tcx.h**. Then reset the Makefile to load its header files from **$(INSTALL_DIR)/include** and recompile. This way you can catch any files that use **tcxP.h** at compile time rather than finding them out through catastrophic and mysterious crashes at run time.

You must also change the Makefile to link from the IPT library as we specified in the previous section.

If you accidentally run a TCX module with the IPT system, the IPT server will die immediately with a core dump. Watch for this symptom and recognize it.

Running the IPT server

The IPT server simply acts as the system "operator." It connects clients together and provides a way to map message strings to message IDs in a system wide consistent way. No data flows through the IPT server, although modules can use the server to log message headers as they send them. The official specification of the IPT server command line is,

```
$(INSTALL_DIR)/bin/iptserver [-d <domain name> ]
                             [ -l <message log file> ]
```

For most situations, calling it with no arguments and no environment variables set will be sufficient. It will print out a banner proclaiming what version of IPT is being used and what machine the server is running on. This machine is important, since in order for IPT clients to connect to the server you have to know what machine this is.

The **-d** option is used to specify a "domain name." The usage of this is explained detail in the section dealing with setting up and using multi-server systems.

As the system progresses the IPT server will print out what modules it is connecting together and when modules connect and disconnect to the server. If the server dies all the modules must be restarted.

If a module has been told to log messages, the IPT server will print out these message headers. The message headers specify what the message type is, where it was coming from and going to, how big it was, and they give a hint to how it was handled. If you do not specify a log file with the **-l** option, the messages are logged to standard out. See the next section for how to turn on message logging on a module by module basis.

Initializing an IPT client

C++ function
Instance
(ipt/ipt.h)

```
IPCommunicator* IPCommunicator::Instance(const char* module_name,
                                         const char* host_name=0)
```

IPCommunicator::Instance chooses the class of IPCommunicator that is relevant and creates it. The client will be called *module_name* and will connect to a IPT server running on the machine *host_name*. If *host_name* is **0,** (its default, since it is an optional argument) then IPT will look for the name of the machine running the IPT server in the environment variable **IPTHOST**. If the **IPTHOST** environment variable is not set, then IPT will assume that the IPT server is running on the same machine as this client.

We implement the initialization this way to make it possible to have different subclasses of **IPCommunicator** that can use different types of connections. **IPCommunicator::Instance** will automatically assess what kind of machine it is running on and choose the appropriate subclass of **IPCommunicator**.

IPCommunicator::Instance is a static member function, which means you invoke it just as it is written without needed to pass in a preexisting **IPCommunicator**. For example,

```
IPCommunicator* comm = IPCommunicator::Instance("Test1");
```

will create an **IPCommunicator** instance with module name *"Test1"* and with the host name passed in as 0, i.e. with the host name gotten from the **IPTHOST** environment variable. *comm*, the resulting **IPCommunicator**, can then be used as the instance for all the non-static member functions of **IPCommunicator.**

When an **IPCommunicator** instance is created, it checks to see if the **IPTLOGGING** environment variable is set. If it is, the module will log all the headers of all messages sent and received by that module with the IPT server.

Another environment variable that is checked for is **IPTOUTPUT**. If **IPTOUTPUT** is unset, or is set to "stdout", then IPT outputs status information to standard out. If **IPTOUTPUT** is "none" then all IPT output is redirected to **/dev/null**. If **IPTOUTPUT** is set to anything else, then it is assumed to be a file name and all IPT output is redirected to that file.

One last environment variable that is relevant is **IPTMACHINE**. If you set this environment variable, then IPT uses the result as the name of the machine. If you do not set this environment variable, IPT uses the standard operating system calls to get the name of the machine.

C function
iptCommunicatorInstance
(ipt/ipt.h)

```
IPCommunicator* iptCommunicatorInstance(char* module_name,
                                        char* host_name)
```

This routine uses **IPCommunicator::Instance** to create the communicator instance and returns it to you.

Getting information about the communicator

C++ Function
ModuleName
(ipt/ipt.h)

```
const char* IPCommunicator::ModuleName()
```

Returns the name of the module as registered in **IPCommunicator::Instance**.

C Function
iptModuleName
(ipt/ipt.h)

```
char* iptModuleName(IPCommunicator* comm)
```

C cover function for **IPCommunicator::ModuleName**.

C++ Function
ServerHostName
(ipt/ipt.h)

```
const char* IPCommunicator::ServerHostName()
```

Returns the name of the machine that the IPT server is running on.

C++ Function
ThisHost
(ipt/ipt.h)

```
const char* IPCommunicator::ThisHost()
```

Returns the name of the machine that the client is running on.

C Function
iptThisHost
(ipt/ipt.h)

```
char* iptThisHost(IPCommunicator* comm)
```

C cover function for **IPCommunicator::ThisHost**.

C++ Function
DomainName
(ipt/ipt.h)

```
const char* IPCommunicator::DomainName()
```

Returns the name of the client's domain. Returns NULL if there is no defined domain name.

C Function
iptDomainName
(ipt/ipt.h)

```
char* iptDomainName(IPCommunicator* comm)
```

C cover function for **IPCommunicator::DomainName**.

Setting up connections

C++ Function
Connect
(ipt/ipt.h)

```
IPConnection* IPCommunicator::Connect(const char* mod_name,
                                      int required=IPT_REQUIRED)
```

This member function requests a connection to a module named *mod_name*. If the module named *mod_name* is already connected then *mod_name*'s connection is returned regardless of the value of *required*.

If *required* is **IPT_REQUIRED** then the function blocks until the server reports that the module named *mod_name* has connected to the server, then the connection between this client and the module named *mod_name* is set up and that connection is returned.

If *required* is set to **IPT_OPTIONAL** then if the module named *mod_name* is already connected to the IPT server, the resulting connection between the client and *mod_name* is returned. If the module named *mod_name* is not connected to the IPT server, then a connection with the proper name is declared and returned, but it will not be active.

If *required* is **IPT_NONBLOCKING**, then if the module named *mod_name* is not already connected to the client an inactive connection is always returned, but when the module named *mod_name* connects to the IPT server then the IPT server will initiate a connection between the client and the module named *mod_name*. This allows us to set up standing request for a connection to a module without having to block until that module comes up.

C Function
iptConnect
(ipt/ipt.h)

```
IPConnection* iptConnect(IPCommunicator* comm,
                        char* mod_name, int required)
```

C cover function for **IPCommunicator::Connect**.

Manipulating connections

C++ Function
Name
(ipt/connection.h)

```
const char* IPConnection::Name()
```

Returns the name of the connected module

C Function
iptConnectionName
(ipt/ipt.h)

```
char* iptConnectionName(IPConnection* conn)
```

C cover function for **IPConnection::Name**.

C++ Function
Host
(ipt/connection.h)

```
const char* IPConnection::Host
```

Returns the name of the machine on which the connected module is running.

C Function
iptConnectionHost
(ipt/ipt.h)

```
char* iptConnectionHost(IPConnection* conn)
```

C cover function for **IPConnection::Host**.

C++ Function
FD
(ipt/connection.h)

```
int IPConnection::FD()
```

Returns the file descriptor associated with the connection. If the connection is inactive, or file descriptors are not applicable to this type of connection, return -1.

C Function
iptConnectionFd
(ipt/ipt.h)

```
int iptConnectionFd(IPConnection* conn)
```

C cover function for **IPConnection::FD**.

C++ Function
Active
(ipt/connection.h)

```
int IPConnection::Active()
```

This might be the most useful of the member functions of IPConnection. You can use it to see if a connection is active or not. If the connection is active that means it can send and receive data and there is a live module on the other end. If this is the case, IPConnection::Active returns 1, if not it returns 0.

C Function
IPConnection::Active
(ipt/ipt.h)

```
int iptConnectionActive(IPConnection* conn)
```

C cover function of **IPConnection::Active**.

Connection/disconnection callbacks

IPT lets you set up callbacks for when modules connect or disconnect. These callbacks can be for a particular module or for any module. In many systems, these callbacks are implemented by pointers to functions. IPT is written in C++, and has a more flexible callback mechanism for connections and disconnections.

To register a connection callback you pass the callback registration routines an instance of a subclass of **IPConnectionCallback**. **IPConnectionCallback** is a abstract virtual class whose only member function is **Execute**:

```
class IPConnectionCallback {
     void Execute(IPConnection* conn);
};
```

The flexibility this gives is the option to create "parameters" without having to pass in data cast through void. For example, say you wanted to have a connection callback that printed a number as a parameter when a connection was made. The connection callback class would look something like:

```
class NumberConnectionCallback : public IPConnectionCallback {
     NumberConnectionCallback(int n) { _n = n; }
     void Execute(IPConnection* conn) { work_with(conn, _n); }
  private:
     int _n;
};
```

Then you would create an instance of **NumberConnectionCallback** with a parameter that would ultimately be dealt with by **work_with** within **Execute**.

IPT provides some convenience mechanisms for defining connection callbacks to invoke a member function of a class. This could be done using templates, but we use some macro magic for backward compatibility with older C++ compilers.

C++ Macro
declareConnectionCallback
(ipt/callbacks.h)

```
declareConnectionCallback(class_name)
```

This C++ macro declares a subclass of **IPConnectionCallback** that can be created with a member function of class *class_name* as a parameter.

C++ Macro
implementConnection Callback
(ipt/callbacks.h)

```
implementConnectionCallback(class_name)
```

This C++ macro implements the code necessary for a subclass of **IPConnectionCallback** that can be created with a member function of class *class_name* as a parameter. **implementConnectionCallback** should only be done once per class, but **declareConnectionCallback** can be done as necessary in different files since it is just the class definition of the **IPConnectionCallback** subclass.

Once a connection callback has been declared and implemented this way, you can create one and associate it with a member function by doing

```
new ConnectionCallback(class_name)(instance,
  &class_name::member_func)
```

class_name is the name of the class you passed into the macros. *instance* is an instance of that class whose member function *member_func* will be invoked when the callback is executed. *member_func* must be of the form,

```
class_name::member_func(IPConnection* conn) ...
```

So, when the callback is executed, **instance->member_func(conn)** is invoked, where *conn* is the connection that the callback is executed for.

Normally, you would create the connection callback inside of another member function of class *class_name*, so the instance passed in would be the this pointer. As an example,

```
class TestClass {
  TestClass::TestClass();
  void connect_callback(IPConnection* c) {
      printf("Something happening with %s\n", c->Name());
};

declareConnectionCallback(TestClass);
implementConnectionCallback(TestClass);

TestClass::TestClass()
{
  IPConnectionCallback* callback;
  callback =
      new ConnectionCallback(TestClass)(this,
      &TestClass::connect_callback);
      .
      .
}
```

C++ Function
AddConnectCallback
(ipt/ipt.h)

```
void IPCommunicator::
        AddConnectCallback(IPConnectionCallback* callback)
```

This member function adds a global connection callback *callback* to the list of callbacks. Whenever any module connects *callback* will be executed.

C++ Function
AddConnectCallback
(by routine)
(ipt/ipt.h)

```
void IPCommunicator::
        AddConnectCallback(void (*func)(IPConnection* conn,
                                         void* data),
                           void* param)
```

This member function provides a shortcut for C++ callback of a function. The function, *func*, is passed the connection, *conn*, that is being activated and *data*, which is set equal to *param*.

C Function
iptAddConnectCallback
(ipt/ipt.h)

```
void iptAddConnectCallback(IPCommunicator* comm,
                           void (*func)(IPCommunicator* comm,
                                        IPConnection* conn,
                                        void* data),
                           void* param)
```

This is a C cover function for **IPCommunicator::AddConnectCallback.** Instead of passing in a callback class, which would be hard to do in C, you should pass in a pointer to a function, *func*. *func* should take three arguments: *comm*, the communicator instance, *conn*, the connection that has become active and *data*, which will be *param*. You can use *param* to communicate structures to the callback routines without needing any global variables.

C++ Function
AddDisconnectCallback
(ipt/ipt.h)

```
void IPCommunicator::
       AddDisconnectCallback(IPConnectionCallback* callback)
```

This member function adds a global connection callback *callback* to the list of callbacks. Whenever any module disconnects *callback* will be executed.

C++ Function
AddDisconnectCallback
(by routine)
(ipt/ipt.h)

```
void IPCommunicator::
       AddDisconnectCallback(void (*func)(IPConnection* conn,
                                          void* data),
                             void* param)
```

This member function provides a shortcut for C++ callback of a function. The function, *func*, is passed the connection, *conn*, that is being deactivated and *data*, which is set equal to *param*.

C Function
iptAddDisconnectCallbac
k
(ipt/ipt.h)

```
void iptAddDisconnectCallback(IPCommunicator* comm,
                              void (*func)(IPCommunicator* comm,
                                           IPConnection* conn,
                                           void* data),
                              void* param)
```

This is a C cover function for **IPCommunicator::AddDisconnectCallback.** Instead of passing in a callback class, which would be hard to do in C, you should pass in a pointer to a function, *func*. *func* should take three arguments: *comm*, the communicator instance, *conn*, the connection that has become active and *data*, which will be *param*. You can use *param* to communicate structures to the callback routines without needing any global variables.

C++ Function
AddConnectCallback
(IPConnection)
(ipt/connection.h)

```
void IPConnection::
     AddConnectCallback(IPConnectionCallback* callback)
```

You can use this member function of IPConnection to add a callback to a particular connection rather than to all connections in general. When the connection becomes active, *callback* is executed.

C Function
iptConnectionAddConnectCallback
(ipt/ipt.h)

```
void iptConnectionAddConnectCallback(IPConnection* conn,
                    void (*func)(IPCommunicator* comm,
                                 IPConnection* c, void* data),
                    void* param)
```

This is a C cover function for **IPConnection::AddConnectCallback**. Instead of a **IPConnectionCallback** handler subclass, it causes the function *func* to be invoked with arguments of the newly active connection and *data*, which is set to *param*.

C++ Function
AddDisconnectCallback (IPConnection)
(ipt/connection.h)

```
void IPConnection::
     AddDisconnectCallback(IPConnectionCallback* callback)
```

You can use this member function of IPConnection to add a callback to a particular connection rather than to all connections in general. When the connection becomes inactive, *callback* is executed.

C Function
iptConnectionAddDisconnectCallback
(ipt/ipt.h)

```
void iptConnectionAddDisconnectCallback(IPConnection* conn,
                    void (*func)(IPCommunicator* comm,
                                 IPConnection* c, void* data),
                    void* param)
```

This is a C cover function for **IPConnection::AddDisconnectCallback**. Instead of a **IPConnectionCallback** handler subclass, it causes the function *func* to be invoked with arguments of the newly inactive connection and *data*, which is set to *param*.

Registering message types

A message type consists of a several basic components. The most basic is the message type name, which is an arbitrary string which should be unique across the system. This message type name will be translated into a message type ID by the central server, and the message ID is an integral part of the message type as well. A message type can also have a message format specifier attached to it. This format specifier starts out as a string which directs how to unpack the raw message data into a C or C++ structure. The code for doing this formatting was mostly inherited from TCX, and the syntax for format specification is in the next section. Under IPT, the format specifier is completely optional, and can be NULL. A message with a NULL format is always dealt with on the raw data level.

Message types also encode things like whether a message is handled and how it is handled. Message types encode "destinations," i.e., what to do with incoming information, whether to put it on a queue or put it in a pigeonhole.

The first step in using message types is to register them. The registration process simply checks with the central server to get a consistent mapping between message names and message ID's. No other information is sent to the server and other message type registration, such as adding handler and destination information is done with later routines.

Each message type is added by name to internal tables which can be used to lookup messages by name or by ID.

C++ Function
RegisterMessage
(ipt/ipt.h>

```
IPMessageType* IPCommunicator::RegisterMessage(constchar*msg_name,
                                               const char* fmt=0)
```

This member function takes a message name *msg_name* and an optional format specification string *fmt* and creates a message type. The message type is registered with the server and added to the internal message type tables.

C Function
iptRegisterMessage
(ipt/ipt.h)

```
IPMessageType* iptRegisterMessage(IPCommunicator* comm,
                                  char* msg_name, char* fmt)
```

C cover function for **IPCommunicator::RegisterMessage**

C++ Function
RegisterMessages
(ipt/ipt.h)

```
void IPCommunicator::RegisterMessages(IPMessageSpec* messages)
```

This member function takes an array of message name/format string specifiers *messages* and registers each one of those messages with the server and the internal message type tables.

IPMessageSpec is a structure that looks like,

```
struct IPMessageSpec {
     char* name;
     char* fmt;
};
```

Where *name* is the message name and *fmt* is the format specifier string (which can be NULL to indicate no format).

The array of **IPMessageSpecs** passed to **RegisterMessages** must be terminated by a **{NULL, NULL}** pair.

One advantage that **RegisterMessages** has over **RegisterMessage** is that it packs all of the registration requests to the server into one message so that it can avoid the overhead of breaking the request into multiple messages.

C Function
iptRegisterMessages
(ipt/ipt.h)

```
void iptRegisterMessages(IPCommunicator* comm,
                         IPMessageSpec* messages)
```

C cover function for **IPCommunicator::RegisterMessages**. **IPMessageSpec** is defined in *<ipt/ipt.h>* in the same way it is defined in *<ipt/ipt.h>*.

Specifying data formats

The user represents format information as a string which is parsed by IPT into an internal format representation. The data format string reassembles a C language **typedef** and specifies the layout of a particular data structure. The data format string may be either a primitive type, a composite type, or a user defined type. Primitive types included in the data format language are:

```
char, short, int, float, double, boolean and string
```

A boolean is defined in C as 0 for FALSE and 1 for TRUE. The type string in C is treated as a NULL terminated ('\0') list of characters.

An example of a primitive data format is:

```
typedef int SAMPLE_INT_TYPE;
#define SAMPLE_INT_FORM "int"
```

Composite data formats are aggregates of other data types. The supported composites include structures, fixed length arrays, variable length arrays, pointers and self referencing pointers.

- Structures are denoted by a pair of braces surrounding a list of data types separated by commas:

```
typedef struct {
     char *str;
     int x;
     int y;
} SAMPLE_STRUCT_TYPE;
#define SAMPLE_STRUCT_FORM "{string, int, int}"
```

- Fixed Length Arrays are denoted by square brackets that enclose a data format specification, a colon, and then a list of one or more dimensions separated by commas:

```
typedef struct {
     int array[17][42];
     char *str;
} FIXED_ARRAY_TYPE;

#define FIXED_ARRAY_FORM "{[int: 17, 42], string}"
```

- Variable Length Arrays are denoted by angle brackets. The data format specification is the same as for fixed length arrays, except that the dimension numbers refer to an element of the enclosing structure that contains the value of the actual dimension. In the **VARIABLE_ARRAY_FORM** example, the "1" refers to the first element of the structure, which contains the number of elements in the variable length array. Variable Length Array formats must be directly embedded in a structure format:

```
typedef struct {
     int arrayLength;
     int* variableArray;
} VARIABLE_ARRAY_TYPE;

#define VARIABLE_ARRAY_FORM "{int, <int: 1>}"
```

Variable length arrays can have more than one dimension specifier. This means that you can specify that the total number of elements in a variable array is gotten from the product of several integer elements of the structure. For example,

```
typedef struct {
     int rows, cols;
     int* matrix;
} VARIABLE_ARRAY_TYPE;
#define MATRIX_ARRAY_FORM "{int, int, <int: 1, 2>}"
```

In this case, the total number of elements in **matrix** is specified by **rows*cols**.

- Pointers are denoted by an * followed by a data format specification. If the pointer value is NULL no data is encoded or sent. Otherwise data is sent and the receiving end creates a pointer to the data. Note that only the data is passed, not the actual pointers, so that structures that share structure or point to themselves (cyclic or doubly linked lists) will not be correctly reconstructed.

```
typedef struct {
     int x, *pointerToInt;
} POINTER_EXAMPLE_TYPE;

#define POINTER_EXAMPLE_FORM "{int, *int}"
```

- The self pointer definition, *!, is used for defining linked or recursive data formats. The self pointer format refers to the enclosing data structure. IPT will translate linked data structures into a linear form before sending them and then recreate the linked form in the receiving module. These routines assume that the end of the linked list is designated by a NULL pointer value. Therefore it is important that all linked data structures be NULL for the encoding routines to work correctly.

```
typedef struct _EXAMPLE {
     int x;
     struct _EXAMPLE *next;
} EXAMPLE_TYPE;

#define EXAMPLE_FORM "{int, *!}"
```

In addition to specifying data as a list of defined primitives, one can also give an integer number specifying the number of bytes to send. In the examples below each data format is equivalent.

```
typedef struct {
     int x;
     int y;
     char *string;
} SAMPLE_TYPE;

#define SAMPLE_DATA_FORM_1 {int, int, string}"
#define SAMPLE_DATA_FORM_2 "{4, 4, string}"
#define SAMPLE_DATA_FORM_3 "{8, string}"
```

IPT allows you to extend the format specification with your own user defined format specifiers.

RegisterNamedFormatter
(ipt/ipt.h)
C++ Function

```
IPFormat* IPCommunicator::RegisterNamedFormatter(
            const char* format_name, const char* format_spec)
```

This method associates the format string *format_spec* with the name *format_name*. *format_name* can then be used as a "primitive" in other format specifications.

iptRegisterNamedFormatter
(ipt/ipt.h)
C Function

```
IPFormat* iptRegisterNamedFormatter(IPCommunicator* comm,
                                    char* format_name,
                                    char* format_spec)
```

C cover function for **IPCommunicator::RegisterNamedFormatter**.

RegisterNamedFormatters
(ipt/ipt.h)
C++ Function

```
void IPCommunicator::RegisterNamedFormatters(
                         IPFormatSpec* formats)
```

This method takes an array of format specifiers, *formats*. The structure **IPFormatSpec** is simply

```
struct IPFormatSpec {
     char* name;
     char* format;
};
```

name should be the name of the format and *format* should be the format specifier itself. The last element of the array should have *name* and *format* set to **NULL** to mark the end of the array.

iptRegisterNamed Formatters
(ipt/ipt.h)
C Function

```
void iptRegisterNamedFormatters(IPFormatSpec* formats)
```

C cover function for **IPCommunicator::RegisterNamedFormatters**.

Looking up message types

Message types can be accessed either by message name or by the ID assigned by the IPT server

C++ Function
LookupMessage (by name)
(ipt/ipt.h)

```
IPMessageType* IPCommunicator::LookupMessage(const char* msg_name)
```

Looks up the message type with the name *msg_name*. If a type with that name has been registered the member function returns it, otherwise it returns NULL.

C Function
iptLookupMessage
(ipt/ipt.h)

```
IPMessageType*iptLookupMessage(IPCommunicator*comm,char*msg_name)
```

C cover function for **IPCommunicator::LookupMessage(const char*)**

C++ Function
LookupMessage (by ID)
(ipt/ipt.h)

```
IPMessageType* IPCommunicator::LookupMessage(int id)
```

Looks up the message type with the ID *id*. If a type has been assigned that ID by the central server this routine will return it, otherwise it will return NULL.

C Function
iptMessageById
(ipt/ipt.h)

```
IPMessageType* iptMessageById(IPCommunicator* comm, int id)
```

C cover function for **IPCommunicator::LookupMessage(int)**.

Registering message handlers

You can associate a message handler with a message type. This means that when a message of that type comes in, instead of going onto a message queue the receipt of the message causes a callback procedure to be invoked.

The callback procedures are specified in the same way for handlers as they were for connections (see "Connection/disconnection callbacks" on page 10 for more general information on callbacks) except the base class for a message handler is **IPHandler-Callback** instead of **IPConnectionCallback**. The definition of **IPHandlerCallback** is

```
class IPHandlerCallback {
     virtual void Execute(IPMessage* message);
};
```

The **Execute** member function gets passed the message that is being handled.

C++ Macro
declareHandlerCallback
(ipt/callbacks.h)

```
declareHandlerCallback(class_name)
```

Declares a subclass of **IPHandlerCallback** that can be used to invoke member functions of class *class_name* in response to messages. For examples of how to use this see "Connection/disconnection callbacks" on page 10.

C++ Macro
implementHandler Callback
(ipt/callbacks.h)

```
implementHandlerCallback(class_name)
```

Implements a subclass of **IPHandlerCallback** that can be used to invoke member functions of class *class_name* in response to messages. For examples of how to use this see "Connection/disconnection callbacks" on page 10.

C++ Function
RegisterHandler
(ipt/ipt.h)

```
void IPCommunicator::RegisterHandler(IPMessageType* type,
                                     IPHandlerCallback* callback,
                                     int context=IPT_HNDL_STD)
```

This is the basic mechanism by which you can pair a message handler with the message type *type*. *context* should declare under what situations a handler can be invoked.

There are three possible contexts. The default is **IPT_HNDL_STD**. Under this context when a message of type *type* comes in the callback specified by callback is executed unless the message came in during the execution of the callback. For example, say a message of type "foo" comes in and is handled eventually by the routine **handle_foo**. If inside **handle_foo** another message of type "foo" comes in the handling of that message is postponed until handle_foo exits. Once **handle_foo** exits it is called again with the postponed message. Under context **IPT_HNDL_ALL**, a handler can be called even when that handler is in the middle of being executed. So, if inside of **handle_foo** a message of type foo comes along, and **handle_foo** invokes an IPT call to query for a message, sleep, or idle, then **handle_foo** will be recursively called. Setting the context to **IPT_HNDL_NONE** means that the handler will never be called, and it has the same effect as irreversibly deactivating the handler.

Invocation of handlers can also be disabled either generally or for a specific message type at any time. Once handlers are enabled then IPT checks the message queues for handleable messages so that they are handled eventually. Message handling will also be disabled inside a **ReceiveMessage** on a particular message type or inside a **Query**. For example, if there is a handler declared for messages of type foo, and you do a query on type foo, that handler will not be invoked. Instead, the message of type foo will be returned as the result of the query. Once the query has finished, the handler is reactivated and reception of messages of type foo will cause the handler to be invoked.

C++ Convenience Function
RegisterHandler
(ipt/ipt.h)

```
void IPCommunicator::
     RegisterHandler(IPMessageType* type,
                    void (*func)(IPCommunicator* comm,
                                 IPMessage* message,
                                 void* data)
                    void* param);
```

This convenience function can be used to avoid having to create any instances of **IPHandlerCallback**. Instead of passing in a subclass of **IPHandlerCallback**, a pointer to the function *func* is passed in along with a parameter, *param*. When a message of type *type* comes in, the function *func* will be invoked given the **IPCommunicator** *comm* that received the message, the message *message* and *data*, which should be equal to *param*.

C Function
iptRegisterHandler
(ipt/ipt.h)

```
void iptRegisterHandler(IPCommunicator* comm, char* msg_name,
                        void (*callback) (IPCommunicator*,
                                          IPMessage*, void*),
                        int context, void* data);
```

This C cover function is similar to the functional **RegisterHandler** convenience member function. It registers the function *callback* as the callback for the message named *msg_name*. The parameter *data* will be the third parameter passed to the function callback when a message named *msg_name* arrives. *context* is the context in which to handle the message.

For your convenience, the structure **IPMsgHandlerSpec** is defined in both *ipt/ipt.h* and *ipt/ipt.h* as

```
typedef struct {
     char* msg_name;
     void (*callback) (IPCommunicator*, IPMessage*, void*);
```

```
    int context;
    void* data;
} IPMsgHandlerSpec;
```

No routine gets passed this structure, but it can be useful for maintaining an array of handler specification information.

Changing handling of messages

C++ Function
DisableHandlers
(ipt/ipt.h)

```
void IPCommunicator::DisableHandlers()
```

Disables the handling of messages of all types.

C Function
iptDisableHandlers
(ipt/ipt.h)

```
void iptDisableHandlers(IPCommunicator* comm);
```

C cover function for **IPCommunicator::DisableHandlers**.

C++ Function
EnableHandlers
(ipt/ipt.h)

```
void IPCommunicator::EnableHandlers()
```

This member function enables the handling of all messages that are not specifically disabled by type. The function checks the message queue and causes all messages with handlers to be handled now.

C Function
iptEnableHandlers
(ipt/ipt.h)

```
void iptEnableHandlers(IPCommunicator* comm);
```

C cover function for **IPCommunicator::EnableHandlers**.

C++ Function
HandlersActive
(ipt/ipt.h)

```
int IPCommunicator::HandlersActive()
```

Returns 0 if the handlers have been disabled by IPCommunicator::DisableHandlers, 1 if not.

C++ Function
DisableHandler
(ipt/ipt.h)

```
void IPCommunicator::DisableHandler(IPMessageType* type)
```

This member function disables the handling of messages of type *type*.

C Function
iptDisableMsgHandler
(ipt/ipt.h)

```
void iptDisableMsgHandler(IPCommunicator* comm, IPMessageType* type)
```

The C cover function for **IPCommunicator::DisableHandler**.

C++ Function
EnableHandler
(ipt/ipt.h)

```
void IPCommunicator::EnableHandler(IPMessageType* type)
```

This member function enables the handling of messages of type *type*. The internal message queue is searched for messages of type *type*. If any messages of type *type* are found, they are removed from the queue and handled.

C Function
iptEnableMsgHandler
(ipt/ipt.h)

```
void iptEnsableMsgHandler(IPCommunicator* comm,
                          IPMessageType* type)
```

The C cover function for **IPCommunicator::EnableHandler**.

C++ Function
HandlerActive
(ipt/messagetype.h)

```
int IPMessageType::HandlerActive()
```

Returns 0 if the handling of this message type has been disabled or the message type has no handler, 1 if there is an active message handler attached to this type.

Creating messages

Generally, you will not have to create messages because the routines provided will create the messages for you, but if you need to create your own messages, IPT provides the necessary facilities.

C++ Creation Function
IPMessage
(raw)
(ipt/message.h)

```
IPMessage::IPMessage(IPMessageType* type,
                     int instance,
                     int size_data, unsigned char* data=0)
```

Every message must be given a type (*type*) and and instance number (*instance*) at creation. Messages also have data. This creation function for **IPMessage** takes the number of bytes that the data should have (*size_data*) and then an optional buffer for the data (*data*).

If *data* is zero, then the creation function allocates *size_data* bytes of data and uses that as the messages data buffer. If *data* is nonzero, then the creation function causes the new message to use *data* as its data buffer with no memory allocation involved. If this is the case, then *data* must point to a buffer of at least length *size_data*.

C++ Creation Function
IPMessage
(formatted)
(ipt/message.h)

```
IPMessage::IPMessage(IPMessageType* type,
                     int instance,
                     int void* data)
```

This creation function assumes that *type* has a data formatter associated with it. It creates a message with the data gotten from the formatted data in *data* as specified by *type*'s data formatter. The message will have instance number *instance*.

C++ Function
generate_instance_num
(ipt/ipt.h)

```
int IPCommunicator::generate_instance_num()
```

Instance numbers passed to messages at creation time can be arbitrary, but you can use this member function to generate a unique instance number. The instance number will be unique for the module, but will not be unique across the system.

C Function
iptNewMessageRaw
(ipt/ipt.h)

```
IPMessage* iptNewMessageRaw(IPMessageType* type, int inst_num,
                            IPConnection* conn,
                            int size_data, unsigned char* data)
```

This C cover function creates a message with type *type*, instance number *inst_num*, connection reference *conn*, and *size_data* bytes of raw data stored in *data*.

C Function
iptNewMessageForm
(ipt/ipt.h)

```
IPMessage* iptNewMessageForm(IPMessageType* type, int inst_num,
                             IPConnection* conn,
                             void* data)
```

This C cover function creates a message with type *type*, instance number *inst_num*, connection reference *conn*, and formatted data stored in *data*.

Example
Making a message

For example, to create a message that contains a string as a raw piece of data, you would do,

```
IPCommunicator* comm = IPCommunicator::Instance("Test");
IPMessageType* type = comm->RegisterMessage("TestMsg");
char* data = "Hello World";
IPMessage* msg =
     new IPMessage(type, comm->generate_instance_num(),
     strlen(data)+1, data);
```

This sample code creates a communicator, uses that communicator to create a message with name "TestMsg" and no format specifier, and then creates a message of type "TestMsg" with the raw unformatted data "Hello World."

Sending messages

C++ Function
SendMessage
(ipt/ipt.h)

```
int IPCommunicator::SendMessage(IPConnection* conn,
                                IPMessage* message)
```

This member function sends a message (*message*) to the connection conn. If it is successful, it returns the number of bytes written. If it is unsuccessful it returns -1. This is true for all the variants of the **SendMessage** member function. **SendMessage** can fail because of *conn* being an inactive connection, or going inactive in the middle of the send, or it might fail simply because of an I/O error.

C++ Convenience Function
SendMessage
(raw)
(ipt/ipt.h)

```
int IPCommunicator::SendMessage(IPConnection* conn,
                                IPMessageType* type,
                                int size, unsigned char* data)
```

This member function creates a message of type *type* and sends that message through the connection conn. The message will contain *size* bytes of unformatted data from buffer *data*.

C++ Convenience Function
SendMessage
(formatted)
(ipt/ipt.h)

```
int IPCommunicator::SendMessage(IPConnection* conn,
                                IPMessageType* type,
                                void* data)
```

This member function creates a message of type *type* and sends that message through the connection conn. The message will contain data gotten from decoding the formatted buffer data according *type*'s format specifier. If *type* does not have a format specifier, the routine will print an error and return -1.

C++ Convenience Function
SendMessage
(raw by name)
(ipt/ipt.h)

```
int IPCommunicator::SendMessage(IPConnection* conn,
                                const char* msg_name,
                                int size, unsigned char* data)
```

This member function creates a message of name *msg_name* and sends that message through the connection conn. The message will contain *size* bytes of unformatted data from buffer *data*. If *msg_name* is not a valid message name, the routine will print an error and return -1.

C Function
iptSendRawMsg
(ipt/ipt.h)

```
int iptSendRawMsg(IPConnection* conn, char* type,
                  int size, unsigned char* data)
```

C cover function for the raw, by message name, **IPCommunicator::SendMessage**. Note that you do not have to pass in an **IPCommunicator**, since that information can be derived from *conn*.

C++ Convenience Function
SendMessage
(formatted by name)
(ipt/ipt.h)

```
int IPCommunicator::SendMessage(IPConnection* conn,
                                const char* msg_name,
                                void* data)
```

This member function creates a message of name *msg_name* and sends that message through the connection conn. The message will contain data gotten from decoding the formatted buffer data according to the type's format specifier. If *msg_name* is not a valid message name or it does not have a format specifier, the routine will print an error and return -1.

C Function
iptSendFormMsg
(ipt/ipt.h)

```
int iptSendFormMsg(IPConnection* conn, char* msg_name,
                   void* data)
```

C cover function for the formatted, by message name, **IPCommunicator::SendMessage**. Note that you do not have to pass in an **IPCommunicator**, since that information can be derived from *conn*.

Receiving messages

An overall note to remember about receiving messages: The convenience functions that let you avoid using the message inspection functions when receiving formatted data can cause a large reduction in efficiency. This reduction is not so drastic for small messages, but for large messages being sent between modules on the same UNIX machine we have measured that receiving an **IPMessage** and then using the **IPMessage** unpacking functions to get at the formatted data can more than double the communications throughput obtained using the convenience functions that receive a message and unpack it for you. The section "Unpacking formatted data from messages" on page 28 explains why this is so.

C++ Function
ReceiveMessage
(ipt/ipt.h)

```
IPMessage* IPCommunicator::
     ReceiveMessage(IPConnection* conn,
                    IPMessageType *type,
                    double timeout = IPT_BLOCK)
```

This member function waits for *time-out* seconds for a message of type *type* from the connection *conn*. If *timeout* is **IPT_BLOCK** (the default), it waits forever for a message matching the description. If *conn* is NULL, any connection will do and if *type* is NULL then any type will do. If *type* is not NULL, message handling for that type is disabled until the end of the call. The routine returns the message received, or NULL if no message has been received or there has been an error. If *conn* is non-NULL and it becomes inactive during the **ReceiveMessage**, the routine will return NULL even if *timeout* is **IPT_BLOCK**.

C++ Function
ReceiveMessage
(by name)
(ipt/ipt.h)

```
IPMessage* IPCommunicator::
     ReceiveMessage(IPConnection* conn,
                    const char* msg_name,
                    double timeout = IPT_BLOCK)
```

This member function is a convenience function that you can pass a message name (*msg_name*) rather than a message type. It works the same way as the basic **ReceiveMessage**.

C Function
iptReceiveMsg
(ipt/ipt.h)

```
IPMessage* iptReceiveMsg(IPCommunicator* comm, IPConnection* conn,
                         char* msg_name, double timeout)
```

C cover function for **IPCommunicator::ReceiveMessage** (by name).

C Function
iptReceiveRawMsg
(ipt/ipt.h)

```
int iptReceiveRawMsg(IPCommunicator* comm,
                     IPConnection* conn, char* msg_name,
                     int max_size, unsigned char* buffer,
                     double timeout)
```

This is a C convenience function to avoid having to deal with **IPMessage**'s. It waits for a message from *conn* of message name *msg_name* according to **IPCommunicator::ReceiveMessage**'s basic rules. If no message is received within *timeout* seconds or there has been an error, it returns 0. If a message matching the criteria is received, it puts up to *max_size* bytes of data from the message into the buffer *buffer*. If the number of bytes received is less than *max_size*, the routine returns the number of bytes placed into *buffer*. If the number of bytes received in the message is greater than *max_size*, *max_size* bytes of the message are placed into *buffer* and -1 is returned.

C++ Function
ReceiveFormatted
(ipt/ipt.h)

```
void* IPCommunicator::ReceiveFormatted(IPConnection* conn,
                                        IPMessageType* type,
                                        double timeout)
```

This C++ member function lets you work directly with the formatted data without having to manipulate messages at all. This convenience does come at the cost of efficiency. Just as in **ReceiveMessage**, the routine waits for timeout seconds for a message through connection *conn* of type *type*. If it receives such a message, it gets the messages formatted data and returns it. The message itself is deleted, but you have to use either **IPCommunicator::DeleteFormatted** or **IPMessageType::DeleteFormatted** to deallocate the data returned since the data will not be associated with any instance of **IPMessage**. If no message matching *conn* and *type* is received within timeout seconds, the routine returns NULL.

C++ Function
ReceiveFormatted
(by name)
(ipt/ipt.h)

```
void* IPCommunicator::ReceiveFormatted(IPConnection* conn,
                                        const char* msg_name,
                                        double timeout)
```

Waits for *timeout* seconds for formatted message named *msg_name* through connection *conn*. If it receives such a message, it returns the data formatted as msg_name's message type specifies. If no such message is received, *msg_name* is not a valid message name, or *msg_name*'s type does not have a format specifier, the routine returns NULL.

C Function
iptReceiveFormMsg
(ipt/ipt.h)

```
void* iptReceiveFormMsg(IPCommunicator* comm,
                        IPConnection* conn, char* msg_name,
                        double timeout)
```

C Cover function for **IPCommunicator::ReceiveFormatted** (by name).

C++ Function
ReceiveFormatted
(into contents)
(ipt/ipt.h)

```
int IPCommunicator::ReceiveFormatted(IPConnection* conn,
                                      IPMessageType* type,
                                      void* msg_contents,
                                      double timeout)
```

This member function allows you to receive a formatted message of type *type* through connection *conn*, but the formatted data is put into the location pointed to by *msg_contents*. The return returns 1 if a message matching *type* and *conn* has been received and 0 if not.

The structure that *msg_contents* points to must be large enough to accept the data, or a memory leak will occur. For example, say you have a format specified by

```
"{ float, float, int, <float : 3 >"
```

The structure that would go with this might look like

```
struct TestType {
     float x, y;
     int num_elem;
     float* elems
};
```

Now if you use **ReceiveFormatted** (into contents) you would have to pass in *msg_contents* as a pointer to a structure of type **TestType** that you have already allocated, i.e.,l

```
TestType t;
if (comm->ReceiveFormatted(conn, type, &t, IPT_BLOCK))
     printf("Received %d elements\n", t.num_elem);
```

Now, while you allocated **t** on the stack, the array **t.elems** has been allocated by IPT. To deallocate all memory associated with a structure filled using this routine you must use either **IPCommunicator::DeleteContents** or **IPMessageType::DeleteContents**, since the formatted data will not be associated with any **IPMessage** instance.

C++ Function
ReceiveFormatted
(into contents, by name)
(ipt/ipt.h)

```
int IPCommunicator::ReceiveFormatted(IPConnection* conn,
                                      const char* msg_name,
                                      void* msg_contents,
                                      double timeout)
```

This convenience member function is used to receive a formatted message of the type named *msg_name* into the structure pointed to by *msg_contents*.

C Function
iptReceiveFormMsg
Contents
(ipt/ipt.h)

```
void* iptReceiveFormMsgContents(IPCommunicator* comm,
                                IPConnection* conn, char* msg_name,
                                void* msg_contents, double timeout)
```

C Cover function for **IPCommunicator::ReceiveFormatted** (into contents, by name).

Inspecting messages

C++ Function
Instance
(ipt/message.h)

```
int IPMessage::Instance()
```

Returns the message's instance number.

C Function
iptMessageInstance
(ipt/ipt.h)

```
int iptMessageInstance(IPMessage* msg)
```

C cover function for **IPMessage::Instance**.

C++ Function
Type
(ipt/message.h)

```
IPMessageType* IPMessage::Type()
```

Returns the message's type.

C Function
iptMessageType
(ipt/ipt.h)

```
IPMessageType* iptMessageType(IPMessage* msg)
```

C cover function for **IPMessage::Type**.

C++ Function
Connection
(ipt/message.h)

```
IPConnection* IPMessage::Connection()
```

If the message has been received from a connection, either by a handler or by **ReceiveMessage**, it will have the connection that it came through associated with it. This member function gives you access to that connection.

C Function
iptMessageConnection
(ipt/ipt.h)

```
IPConnection* iptMessageConnection(IPMessage* msg)
```

C cover function for **IPMessage::Connection**

C++ Function
SizeData
(ipt/message.h)

```
int IPMessage::SizeData()
```

Returns the number of unformatted bytes in the message buffer.

C Function
iptMessageSizeData
(ipt/ipt.h)

```
int iptMessageSizeData(IPMessage* msg)
```

C cover function for **IPMessage::SizeData**.

C++ Function
Data
(ipt/message.h)

```
unsigned char* IPMessage::Data()
```

Returns a pointer to the unformatted message data buffer.

C Function
iptMessageData
(ipt/ipt.h)

```
unsigned char* iptMessageData(IPMessage* msg)
```

C cover function for **IPMessage::Data**.

C++ Function
Print
(ipt/message.h)

```
int IPMessage::Print(int print_data = 0)
```

This method will always print the message header, i.e., the message type, the message ID, the message connection name (if any), and the size of the data held by the message. If you pass in 1, then it tries to print the message data as specified by the message type's formatter, if any. The method returns 1 if it successfully printed the formatted data, and 0 if not.

C Function
iptMessagePrint
(ipt/ipt.h)

```
void iptMessagePrint(IPMessage* msg, int print_data)
```

C cover function for **IPMessage::Print**.

Unpacking formatted data from messages

Instances of **IPMessage** have associated with them raw data. This is simply the bytes that came straight off of the connection. Often, to make this data useful you need to format it. Formatting takes the raw data and puts it into a C structure that you can read and manipulate, and it insures that data will be in the proper format for your platform, even if the format for the equivalent C structure on a different platform is different, i.e., the formatters will fix the byte order of integers to match the byte order of your platform.

C++ Function
FormattedData
(ipt/message.h)

```
void* IPMessage::FormattedData(int force_copy = 0)
```

Uses the messages format specifier to create the formatted data from the internal unformatted message data. The routine returns a pointer to the allocated formatted data. If *force_copy* is 1, calling this member function can cause the raw data to change and IPT avoids as many memory allocations as possible by using pointers into the raw data and doing formatting operations such as changing byte order in place. With *force_copy* 1, this method function also associates the formatted data with the message, so that when the message is deleted, the formatted data is deleted as well. If *force_copy* is 1, what is returned has no association with the message that was formatted, and one of the formatted data deletion functions should be called on the result to deallocate memory. Because of the additional memory allocation and copying involved in making a copy, forcing a copy can much more time than formatting in place and associating the formatted data with the message.

C Function
iptMessageFormData
(ipt/ipt.h)

```
void* iptMessageFormData(IPMessage* msg)
```

C cover function for **IPMessage::FormattedData**, without forced copying

C Function
iptMessageFormCopy
(ipt/ipt.h)

```
void* iptMessageFormCopy(IPMessage* msg)
```

C cover function for **IPMessage::FormattedData**, with forced copying

C++ Function
FormattedData (contents)
(ipt/message.h)

```
void IPMessage::FormattedData(void* data, int force_copy = 0)
```

Uses the messages format specifier to create and copy the formatted data from the internal unformatted message data into *data*. *data* must be a data structure large enough to accommodate the message's formatted data, or problems will ensue. While the member function does not allocate memory for the top level of the formatted data, since that is already done by the user passing in *data*, it does allocate all memory for any substructures, such as variable arrays and strings.

If *force_copy* is 1, then what is copied into data has no relation to the message. If *force_copy* is 0, then the amount of memory allocation and copying is minimized by using chunks of the raw data and doing any data conversion in place. If *force_copy* is 1, then one of the formatted data contents deleters must be called on data.

It is not true that the contents formatting routines to get at formatted data will always be more efficient, than the regular formatted data access routines. In fact, there will be many situations where the regular formatted data access routines will be much more efficient. Basically, if *force_copy* is 0 and the structure that is being formatted contains no pointers and has no needed alignment adjustments (i.e., for a Sparc architecture, it contains only elements of size of integer), then the regular formatting process is just a no-op, i.e. it just returns a pointer to the raw data. The contents formatting procedures have to copy into the structure you provide.

Also note that if you use this member function (or its C equivalent) inside of a message handler, you are responsible for deleting the memory before the handler returns. If you use the other **FormattedData** member function or its equivalent, or use this member function with forced copying, IPT will take care of deallocating the memory associated with the message.

C Function
iptMessageFormContents
(ipt/ipt.h)

```
void iptMessageFormContents(IPMessage* msg, void* data)
```

C cover function for **IPMessage::FormattedData (contents)** without forced copying

C Function
iptMessageForm ContentsCopy
(ipt/ipt.h)

```
void iptMessageFormContentsCopy(IPMessage* msg, void* data)
```

C cover function for **IPMessage::FormattedData (contents)** with forced copying.

Deleting messages and message data

You the user are responsible for much of the memory management for IPT. Any messages that you receive outside of handlers must be deleted. IPT will take care of deleting messages that are passed into message handlers. Formatted data gotten from the messages no longer necessarily has to be deleted by the users, since the messages themselves hold pointers to the formatted data. Deleting messages will take care of the unformatted data and the formatted data that you might have created along the way and associated with that message. Care must be taken with using the "contents" data formatting routines. For the deletion of the formatted contents to work properly, the life-span of the structure you pass in to the contents formatting routines must be longer than the life-span of the message. For example, if you receive a message inside a function, and parse its formatted data into the structure **ParsedData** which is on the stack, you must either explicitly use **IPMessage::DeleteContents** to delete the contents of **ParsedData** or delete the message itself before the function exits, or the internal pointer that the message maintains for the data structure will become invalid.

In C++, deleting a message is a simple matter of including *<ipt/message.h>* using the destructor, i.e.,

```
IPMessage* msg = comm->ReceiveMessage()
delete msg;
```

This will free the memory associated with the message, *msg*. If the message was created with user data, then that data is not freed. If the message created its own data buffer, that data buffer is freed. If the message had formatted data associated with it, that is deleted as well.

We provide additional functions to delete formatted data and to do the memory management from C. The deletion functions that are member functions of the **IPMessage** class (or C covers for these member functions) can be called on any formatted data produced by that message. The following example would work no matter what the value of **force_copy** was.

```
IPMessage* msg = ...;
void* formatted_data = msg->FormattedData(force_copy);
msg->DeleteFormatted(void);
```

The formatted data deletion functions that are not member functions of the **IPMessage** class can only be used on formatted data that is not explicitly associated with an **IPMessage**. The following example will only work if **force_copy** is 1, otherwise we will end up with unpredictable, virtually untraceable bugs.

```
IPCommunicator* comm = ...;
IPMessage* msg = ...; // is of type "TestMsgType"
void* formatted_data = msg->FormattedData(force_copy);
comm->DeleteFormatted("TestMsgType", data);
```

C Function
iptMessageDelete
(ipt/ipt.h)

```
void iptMessageDelete(IPMessage* msg)
```

C cover function for **IPMessage**'s destructor function.

C++ Function
DeleteFormatted
(ipt/message.h)

```
void IPMessage::DeleteFormatted(void* data)
```

If you created formatted data *data* with **IPMessage::FormattedData**, then you can free the memory associated with it with this member function. This function is obsolete, and is included for backwards compatibility.

C Function
iptMessageDeleteForm
(ipt/ipt.h)

```
void iptMessageDeleteForm(IPMessage* msg, void* data)
```

C cover function for **IPMessage::DeleteFormatted**.

C++ Function
DeleteContents
(ipt/message.h)

```
void IPMessage::DeleteContents(void* data)
```

If you created formatted data *data* with **IPMessage::FormattedData (contents)**, then you free the memory associated with it with this member function. In most cases, this function is obsolete, but it can be useful if you want the life-span of the message to be longer than the life-span of *data*.

C Function
iptMessageDelete Contents
(ipt/ipt.h)

```
void iptMessageDeleteContents(IPMessage* msg, void* data)
```

C cover function for **IPMessage::DeleteContents**.

C++ Function
DeleteFormatted
(ipt/messagetype.h)

```
void IPMessageType::DeleteFormatted(const char* msg_name,
                                     void* data)
```

This routine frees the memory associated with *data* and *data* itself as specified by the type's format specifier. This must only be called on data produced by **IPMessage::FormattedData** with forced copying, or you will risk awful, arbitrary, and virtually untraceable bugs.

C++ Function
DeleteContents
(ipt/messagetype.h)

```
void IPMessageType::DeleteContents(void* data)
```

This routine will free the memory associated with *data*, but not *data* itself, as specified by the type's format specifier. This must only be called on data produced by **IPMessage::FormattedData (contents)** with forced copying, or you will risk awful, arbitrary, and virtually untraceable bugs.

C++ Function
DeleteFormatted
(ipt/ipt.h)

```
void IPCommunicator::DeleteFormatted(const char* msg_name,
                                     void* data)
```

This routine looks up the message type named *msg_name*, and then frees the memory associated with *data* and *data* itself as specified by the type's format specifier. This must only be called on data produced by **IPMessage::FormattedData** with forced copying, or you will risk awful, arbitrary, and virtually untraceable bugs.

C Function
iptFreeData
(ipt/ipt.h)

```
void iptFreeData(IPCommunicator* comm, char* msg_name, void* data)
```

C cover function for **IPCommunicator::DeleteFormatted**.

C++ Function
DeleteContents
(ipt/ipt.h)

```
void IPCommunicator::DeleteContents(const char* msg_name,
                                    void* data)
```

This routine looks up the message type named *msg_name*, and then frees the memory associated with *data*, but not *data* itself, as specified by the type's format specifier. This must only be called on data produced by **IPMessage::FormattedData (contents)** with forced copying, or you will risk awful, arbitrary, and virtually untraceable bugs.

C Function
iptFreeContents
(ipt/ipt.h)

```
voidiptFreeContents(IPCommunicator*comm,char*msg_name,void*data)
```

C cover function for **IPCommunicator::DeleteContents**.

Querying

C++ Function
Query
(ipt/ipt.h)

```
IPMessage* IPCommunicator::Query(IPConnection* conn,
                                 IPMessage* query_msg,
                                 IPMessageType* reply_type,
                                 double timeout = IPT_BLOCK)
```

This routine sends the message *query_msg* to the connection *conn*, and then waits for a reply of type *reply_type* with the same message instance as *query_msg*. The routine will wait for *timeout* seconds, or forever if *timeout* is **IPT_BLOCK** (the default). The routine returns the received reply, or NULL if *conn* has gone inactive, there has been an I/O error, or the timeout has expired.

C++ Convenience Function
Query
(raw)
(ipt/ipt.h)

```
IPMessage* IPCommunicator::Query(IPConnection* conn,
                                 IPMessageType* query_type
                                 int size, unsigned char* data,
                                 IPMessageType* reply_type,
                                 double timeout = IPT_BLOCK)
```

This convenience function creates a query message of the type *query_type* and with the *size* bytes of unformatted data *data*.

C++ Convenience Function
Query
(formatted)
(ipt/ipt.h)

```
IPMessage* IPCommunicator::Query(IPConnection* conn,
                                 IPMessageType* query_type,
                                 void* data,
                                 IPMessageType* reply_type,
                                 double timeout = IPT_BLOCK)
```

This convenience function creates a query message of the type *query_type* and with the formatted data *data*. The routine returns NULL if *query_type* has no format specifier.

C++ Convenience Function
Query
(raw, by name)
(ipt/ipt.h)

```
IPMessage* IPCommunicator::Query(IPConnection* conn,
                                 const char* query_name,
                                 int size, unsigned char* data,
                                 const char* reply_name,
                                 double timeout = IPT_BLOCK)
```

This convenience function creates a query message with the name *query_name* and with the *size* bytes of unformatted data *data*. It waits for a message with the name *reply_name*. If either *query_name* or *reply_name* are not valid message names, it returns NULL.

C Function
iptQueryMsgRaw
(ipt/ipt.h)

```
IPMessage* iptQueryMsgRaw(IPConnection* conn,
                          char* query_name,
                          void* data, char* reply_name)
```

C cover function for **IPCommunicator::Query (raw, by name)**. Note that for this C cover function, the timeout is always **IPT_BLOCK**. Also note that you do not have to pass in an instance of **IPCommunicator**, since that can be derived from *conn*.

C++ Convenience Function
Query
(formatted, by name)
(ipt/ipt.h)

```
IPMessage* IPCommunicator::Query(IPConnection* conn,
                                 const char* query_name,
                                 void* data,
                                 const char* reply_name,
                                 double timeout = IPT_BLOCK)
```

This convenience function creates a query message with the name *query_name* and with the formatted data *data*. It waits for a message with the name *reply_name*. If either *query_name* or *reply_name* are not valid message names, it returns NULL.

C Function
iptQueryMsgForm
(ipt/ipt.h)

```
IPMessage* iptQueryMsgForm(IPConnection* conn,
                           char* query_name,
                           void* data, char* reply_name)
```

C cover function for **IPCommunicator::Query (formatted, by name)**. Note that for this C cover function, the timeout is always **IPT_BLOCK**. Also note that you do not have to pass in an instance of **IPCommunicator**, since that can be derived from *conn*.

C++ Convenience Function
QueryFormatted
(ipt/ipt.h)

```
void* IPCommunicator::QueryFormatted(IPConnection* conn,
                                     const char* query_name,
                                     void* data,
                                     const char* reply_name,
                                     double timeout = IPT_BLOCK)
```

This is a convenience C++ member function to avoid having to manipulate **IPMessage**'s. If no reply is received, it returns NULL. If a **IPMessage** is received, it extracts the formatted data from it and returns that formatted data. The original **IPMessage** is deleted.

C Function
iptQueryForm
(ipt/ipt.h)

```
void* iptQueryForm(IPConnection* conn, char* query_name,
                   void* data, char* reply_name)
```

C cover function for **IPCommunicator::QueryFormatted**.

C++ Convenience Function
QueryFormatted
(into contents)
(ipt/ipt.h)

```
int IPCommunicator::QueryFormatted(IPConnection* conn,
                                   const char* query_name,
                                   void* data,
                                   const char* reply_name,
                                   void* reply_data,
                                   double timeout = IPT_BLOCK)
```

This is a convenience C++ member function to avoid having to manipulate **IPMessage**'s. If no reply is received, it returns 0. If a **IPMessage** is received, it extracts the formatted data from it and puts the contents of that data into *reply_data* and returns 1. The original **IPMessage** is deleted.

C Function
iptQueryFormContents
(ipt/ipt.h)

```
int iptQueryFormContents(IPConnection* conn,
                         char* query_name, void* data,
                         char* reply_name, void* reply_data)
```

C cover function for **IPCommunicator::QueryFormatted (into contents)**.

C Function
iptQueryMsgFormTO
(ipt/ipt.h)

```
IPMessage* iptQueryMsgFormTO(IPConnection* conn,
                             char* query_name,
                             void* data, char* reply_name,
                             double timeout)
```

C Function
iptQueryMsgRawTO
(ipt/ipt.h)

```
IPMessage* iptQueryMsgRawTO(IPConnection* conn,
                            char* query_name,
                            void* data, char* reply_name,
                            double timeout)
```

C Function
iptQueryMsgFormTO
(ipt/ipt.h)

```
IPMessage* iptQueryMsgFormTO(IPConnection* conn,
                             char* query_name,
                             void* data, char* reply_name,
                             double timeout)
```

C Function
iptQueryFormContentsTO
(ipt/ipt.h)

```
int iptQueryFormContentsTO(IPConnection* conn,
                           char* query_name, void* data,
                           char* reply_name, void* reply_data,
                           double timeout)
```

These four C cover functions allow a C user to do queries with a timeout of timeout seconds. If the timeout expires before the desired message arrives, then the routines will return NULL or 0 as appropriate.

Replying

If a message comes in from connection A and has instance number I, replying to that message means sending a message (probably with a different type, and almost definitely with different data) to connection A with instance number I. You could construct this reply message yourself using the **IPMessage** constructor, but IPT provides functions which give you a convenient way to respond to queries

C++ Function
Reply
(raw)
(ipt/ipt.h)

```
void IPCommunicator::Reply(IPMessage* msg,
                           IPMessageType* reply_type,
                           int size, unsigned char* data)
```

This member function creates a message of type *reply_type* and with *size* bytes of unformatted data in the buffer *data* and with the same instance as *msg* and sends it to the connection that *msg* was received from.

C++ Function
Reply
(formatted)
(ipt/ipt.h)

```
void IPCommunicator::Reply(IPMessage* msg,
                           IPMessageType* reply_type,
                           void* data)
```

This member function creates a message of type *reply_type* and with formatted data *data* with the same instance as *msg* and sends it to the connection that *msg* was received from.

C++ Function
Reply
(raw, by name)
(ipt/ipt.h)

```
void IPCommunicator::Reply(IPMessage* msg,
                           const char* reply_name,
                           int size, unsigned char* data)
```

This member function creates a message with name *reply_name* and with *size* bytes of unformatted data in the buffer *data* and with the same instance as *msg* and sends it to the connection that *msg* was received from.

C Function
iptReplyRaw
(ipt/ipt.h)

```
void iptReplyRaw(IPCommunicator* comm, IPMessage* msg,
                 const char* reply_name,
                 int size, unsigned char* data)
```

This is a cover function for **IPCommunicator::Reply (raw, by name)**.

C++ Function
Reply
(formatted, by name)
(ipt/ipt.h)

```
void IPCommunicator::Reply(IPMessage* msg,
                           const char* reply_name,
                           void* data)
```

This member function creates a message with name *reply_name* and with formatted data *data* with the same instance as *msg* and sends it to the connection that *msg* was received from.

C Function
iptReplyForm
(ipt/ipt.h)

```
void iptReplyForm( IPMessage* msg,
                 const char* reply_name, void* data)
```

This is a cover function for **IPCommunicator::Reply (formatted, by name)**.

Pigeon-holing message types

There are many situations in a mobile robot system in which you only are interested in the most recent message from any one module. For example, if an asynchronous perception module such as a road follower sends three driving commands to a command arbiter, the arbiter is only really interested in the most recent one.

This could be implemented by "clearing" message queues and putting ReceiveMessage commands in loops with time-outs of 0, but IPT gives a much more efficient mechanism for getting this effect: pigeon-holed messages.

A pigeon hole is a box in which only one message can fit. If another message comes in, the original message is knocked out and replaced by the newer message. IPT's standard pigeon hole mechanism can be thought of as a pigeon hole for each connection for each message type. IPT implements this with the minimum amount of list processing and memory allocation and deallocation.

C++ Function
PigeonHole
(ipt/ipt.h)

```
void IPCommunicator::PigeonHole(IPMessageType* type)
```

This member function declares the message type *type* to be a "pigeonhole." This means that whenever you receive a message of this type, whether by handler or as the result of a **ReceiveMessage**, you are guaranteed to have the most recent message of that type available.

C Function
iptPigeonHoleType
(ipt/ipt.h)

```
void iptPigeonHoleType(IPCommunicator* comm, IPMessageType* type)
```

C cover function for **IPCommunicator::PigeonHole**

C++ Function
PigeonHole
(by name)
(ipt/ipt.h)

```
void IPCommunicator::PigeonHole(const char* msg_name)
```

This member function declares the message type named *msg_name* to be a pigeonhole. If *msg_name* is not a registered message name, the routine prints an error and does nothing.

C Function
iptPigeonHole
(ipt/ipt.h)

```
void iptPigeonHole(IPCommunicator* comm, char* msg_name)
```

C cover function for **IPCommunicator::PigeonHole (by name)**.

Processing loops

Processing of handlers and connecting and disconnecting modules only happens in certain IPT calls. These include **Query** and **ReceiveMessage**. Sometimes you will want to have the program go into a loop for a period of time, perhaps forever, in which it will do nothing except process handlers and manage connection and disconnection. You cannot do this simply by invoking the UNIX command **sleep**, because **sleep** will completely ignore all incoming input. IPT provides you with several commands to "sleep" safely.

C++ Function
MainLoop
(ipt/ipt.h)

```
void IPCommunicator::MainLoop()
```

This member function will loop forever, handling messages and managing connections. It never exits.

C Function
iptMainLoop
(ipt/ipt.h)

```
void iptMainLoop(IPCommunicator* comm)
```

C cover function for **IPCommunicator::MainLoop**.

C++ Function
Sleep
(ipt/ipt.h)

```
int IPCommunicator::Sleep(double time = IPT_BLOCK)
```

This member function will loop for *time* seconds. It exits when time seconds are up, and returns 1 if events were handled during the sleep and 0 if not.

C Function
iptSleep
(ipt/ipt.h)

```
int iptSleep(IPCommunicator* comm, double time)
```

C cover function for **IPCommunicator::Sleep**.

C++ Function
Idle
(ipt/ipt.h)

```
int IPCommunicator::Idle(double time = IPT_BLOCK)
```

This member function will loop for *time* seconds or until an event, such as a message handling, module connection, or disconnection happens. It returns 1 if an event happened in those time seconds and 0 if it just timed out.

C Function
iptIdle
(ipt/ipt.h)

```
int iptIdle(IPCommunicator* comm, double time)
```

C cover function for **IPCommunicator::Idle**.

Client/server relationships

One of the common relationships between modules in an IPT system will be the client/server relationship. In this relationship, the server module has something that many clients want. The server could be a place to query for information of a certain type or just a repository for information produced by the clients.

IPT has some facilities to ease the creation and maintenance of client/server relationships. One module can declare itself a server. Clients then use the IPT client/server code to register with the server. IPT then maintains this relationship throughout the life time of the system, i.e., if the server goes down and comes back up again, the clients will try to reconnect and re-register with the server. Similarly, if clients go down and come up they are deleted and re-added to the server's internal list of active clients. What communications actually happen between clients and servers is up to the system designer.

C++ Function
Server
(ipt/ipt.h)

```
void IPCommunicator::Server(IPMessageType* reg_type = NULL,
                            IPHandlerCallback* callback = NULL)
```

This member function declares the module to be a server. A server needs a registration message, and you can set this registration message to be of type *reg_type*. If *reg_type* is NULL (the default), then the server will be registered with an internal default registration message. When a client registers with the server, the callback *callback* will be invoked. If it is NULL (the default), then no callback is invoked upon registration. Note that any handler registered beforehand for the type *reg_type* will be overridden regardless of whether or not you pass in a callback. In addition, adding your own handler to *reg_type* after you execute the Server will result in unpredictable behavior.

C++ Function
Server
(by name)
(ipt/ipt.h)

```
void IPCommunicator::Server(char* reg_name,
                            IPHandlerCallback* callback = NULL)
```

This is a convenience function for declaring a module a server with registration message of name *reg_name*.

C Function
iptServer
(ipt/ipt.h)

```
void iptServer(IPCommunicator* comm, char* reg_name,
               void (*callback)(IPMessage* msg, void* data),
               void* param)
```

This C cover function declares the module to be a server with a registration message named *reg_name*. If *callback* is non-NULL, IPT calls it with the registration message and *param* as parameters when a client sends a registration message.

C++ Function
Client
(ipt/ipt.h)

```
void IPCommunicator::Client(IPConnection* server,
                            IPMessage* reg_msg = NULL)
```

This member function sends a registration message to server through the connection *server* in order to become a client. If *reg_msg* is NULL (the default), the default registration message is sent. Otherwise, *reg_msg* is the registration message that is sent. *reg_msg* must be of the same type as the server declares in its **IPCommunicator::Server** call, or unpredictable things will happen.

C++ Function
Client
(by name)
(ipt/ipt.h)

```
void IPCommunicator::Client(const char* server_name,
                            IPMessage* reg_msg = NULL)
```

This convenience function attempts to connect to the module named **server_name** and to register this module as a client using either *reg_msg*, or if it is NULL (the default), using the default registration message.

C Function
iptClient
(ipt/ipt.h)

```
void iptClient(IPCommunicator* comm, char* server_name)
```

This C cover function declares this module to be a server of the module named server_name and registers with the default server registration message.

C Function
iptClientRaw
(ipt/ipt.h)

```
void iptClientRaw(IPCommunicator* comm,
                  char* server_name, char* msg_name,
                  int size, unsigned char* data)
```

This C cover function declares this module to be a server of the module named *server_name* and registers with a message named *msg_name* with *size* bytes of unformatted data in buffer *data*.

C Function
iptClientForm
(ipt/ipt.h)

```
void iptClientForm(IPCommunicator* comm,
                   char* server_name, char* msg_name,
                   void* data)
```

This C cover function declares this module to be a server of the module named *server_name* and registers with a message named *msg_name* with formatted data *data*.

C++ Function
Broadcast
(ipt/ipt.h)

```
void IPCommunicator::Broadcast(IPMessage* msg)
```

This member function lets a server broadcast the message *msg* to all of its active clients.

C++ Function
Broadcast
(raw)
(ipt/ipt.h)

```
void IPCommunicator::Broadcast(IPMessageType* type,
                               int size, unsigned char* data)
```

This member function creates a message of type *type* with *size* bytes of unformatted data in the buffer *data* and lets a server broadcast that message to all of its active clients.

C++ Function
Broadcast
(formatted)
(ipt/ipt.h)

```
void IPCommunicator::Broadcast(IPMessageType* type,
                               void* data)
```

This member function creates a message of type *type* with formatted data in the buffer *data* and lets a server broadcast that message to all of its active clients.

C++ Function
Broadcast
(raw, by name)
(ipt/ipt.h)

```
void IPCommunicator::Broadcast(const char* msg_name,
                               int size, unsigned char* data)
```

This member function creates a message with name *msg_name* with *size* bytes of unformatted data in the buffer *data* and lets a server broadcast that message to all of its active clients.

C Function
iptBroadcastRaw
(ipt/ipt.h)

```
void iptBroadcastRaw(IPCommunicator* comm, char* msg_name,
                     int size, unsigned char* data)
```

C cover function for **IPCommunicator::Broadcast (raw, by name)**.

C++ Function
Broadcast
(formatted, by name)
(ipt/ipt.h)

```
void IPCommunicator::Broadcast(const char* msg_name,
  void* data)
```

This member function creates a message with name *msg_name* with formatted data in the buffer *data* and lets a server broadcast that message to all of its active clients.

C Function
iptBroadcastForm
(ipt/ipt.h)

```
void iptBroadcastForm(IPCommunicator* comm,
                      char* msg_name, void* data)
```

C cover function for **IPCommunicator::Broadcast (formatted, by name)**.

Publisher/subscriber relationships

Sometimes, a client/server relationship is not specific enough. IPT also supports a publisher/subscriber relationship between modules. This means that a module can declare that it is a "publisher" for a certain message type. Other modules, the subscribers, can connect to the publisher and subscribe to those message types. Then, when the publisher decides to publish new data, it can publish its message type and only the subscribers to that message type will get the new message.

IPT makes sure that when publishers and clients go up and down, the publisher/client relationship is not broken. For example, when a publisher goes down and comes back up, all of its clients reconnect and resubscribe.

This can be useful for modules that produce data, but which don't want to know where it goes. For example, an obstacle map builder can use this system to "broadcast" its knowledge about obstacles. The "broadcast" is only going to modules that know to subscribe to the obstacle map on the obstacle map builder.

When necessary IPT allows clients to "unsubscribe" from a publisher's information. This allows modules to subscribe for a short time for the information that they need and not burden the system with unnecessary communication during the times that the information is not needed.

C++ Function
DeclareSubscription
(ipt/ipt.h)

```
void IPCommunicator::DeclareSubscription(IPMessageType* type,
                                  IPConnectionCallback*callback=0)
```

The publisher uses this routine to declare a message type type for subscription. Clients can now connect to the publisher and subscribe to this message type. If *callback* is non-NULL (it defaults to NULL), then when a subscriber subscribes to this message type, *callback* is invoked.

C++ Function
DeclareSubscription
(by name)
(ipt/ipt.h)

```
void IPCommunicator::DeclareSubscription(char* msg_name,
                               IPConnectionCallback* callback = 0)
```

This convenience function declares a subscription based on the message named *msg_name*.

C Function
iptDeclareSubscription
(ipt/ipt.h)

```
void iptDeclareSubscription(IPCommunicator* comm, char* reg_name,
                          void(*callback)(IPConnection*subscriber,
                                          void* data),
                          void* param)
```

This C cover function declares the module as a publisher of the message type named *reg_name*. When a module subscribes, *callback* is invoked with the connection subscriber module and *param* as parameters.

C++ Function
Publish
(ipt/ipt.h)

```
void IPCommunicator::Publish(IPMessageType* type, IPMessage* msg)
```

This member function publishes the message *msg* to all of the subscribers to the message type *type*. Note that msg does not have to be of type *type*.

C++ Function
Publish
(raw)
(ipt/ipt.h)

```
void IPCommunicator::Publish(IPMessageType* type,
                          int size, unsigned char* data)
```

This member function creates a message of type *type* with *size* bytes of unformatted data in the buffer *data* and publishes that message to all of the subscribers to *type*.

C++ Function
Publish
(formatted)
(ipt/ipt.h)

```
void IPCommunicator::Publish(IPMessageType* type,
                          void* data)
```

This member function creates a message of type *type* with formatted data in the buffer *data* and publishes that message to all of the subscribers to *type*.

C++ Function
Publish
(raw, by name)
(ipt/ipt.h)

```
void IPCommunicator::Publish(const char* msg_name,
                             int size, unsigned char* data)
```

This member function creates a message with name *msg_name* with *size* bytes of unformatted data in the buffer *data* and publishes that message to all of the subscribers to *msg_name*.

C Function
iptPublishRaw
(ipt/ipt.h)

```
void iptPublishRaw(IPCommunicator* comm, char* msg_name,
                   int size, unsigned char* data)
```

C cover function for **IPCommunicator::Publish (raw, by name)**.

C++ Function
Publish
(formatted, by name)
(ipt/ipt.h)

```
void IPCommunicator::Publish(const char* msg_name,
                             void* data)
```

This member function creates a message with name *msg_name* with formatted data in the buffer *data* and publishes that message to all of the subscribers of *msg_name*.

C Function
iptPublishForm
(ipt/ipt.h)

```
void iptPublishForm(IPCommunicator* comm, char* msg_name, void* data)
```

C cover function for **IPCommunicator::Publish (formatted, by name)**.

C++ Function
Subscribe
(ipt/ipt.h)

```
void IPCommunicator::Subscribe(IPConnection* publisher,
                               IPMessageType* msg_type)
```

This routine is used by a subscriber to subscribe to the message type *msg_type* published by a module connected by *connection* publisher.

C++ Function
Subscribe
(by name)
(ipt/ipt.h)

```
IPConnection* IPCommunicator::Subscribe(const char* pub_name,
                                        const char* msg_name)
```

This connects to a publisher named *pub_name* and subscribes to a message type named *msg_name*. The function returns the established connection with the publisher.

C Function
iptSubscribe
(ipt/ipt.h)

```
IPConnection* iptSubscribe(IPCommunicator* comm, char* pub_name,
                           char* msg_name)
```

C cover function for **IPCommunicator::Subscribe (by name)**.

C++ Function
SubscribeConnection
(ipt/ipt.h)

```
void IPCommunicator::SubscribeConnection(IPConnection* conn,
                                         IPMessageType* msg_type)
```

Used by the publisher to subscribe the module connected to by *conn* to the message type *msg_type*.

C++ Function
SubscribeConnection
(by name)
(ipt/ipt.h)

```
IPConnection* IPCommunicator::SubscribeConnection(const char* name,
                                                  const char* msg_name)
```

Connects to the module named *name* and subscribes it to the message named *msg_name*.

C Function
iptSubscribeConnection
(ipt/ipt.h)

```
IPConnection* iptSubscribeConnection(IPCommunicator*comm,char*name,
                                      char* msg_name)
```

C cover function for **IPCommunicator::SubscribeConnection (by name)**.

C++ Function
Unsubscribe
(ipt/ipt.h)

```
void Unsubscribe(IPConnection* conn, IPMessageType* type)
```

Member function that unsubscribes from the message type *type* being produced by the module connected through *conn*.

C++ Function
Unsubscribe (by name)
(ipt/ipt.h)

```
void Unsubscribe(const char* pub_name, const char* type_name)
```

Member function that unsubscribes from the message type named *pub_name* being produced by the module named *type_name*.

C Function
iptUnsubscribe
(ipt/ipt.h)

```
void iptUnsubscribe(IPCommunicator* comm, char* pub_name,
                     char* type_name)
```

C cover function for **IPCommunicator::Unsubscribe (by name)**.

Example

A persistent system developer could use IPT's publisher/subscriber facilities for doing communications with "wiretaps." The idea of a wiretap is to anonymously "snoop" a connection for messages for debugging. If instead of doing,

```
IPConnection* conn = comm->Connect("friend");
comm->SendMessage(conn,"TheMsg",12,(unsignedchar*)"Helloworld");
```

We do,

```
comm->DeclareSubscription("TheMsg");
comm->SubscribeConnection("friend", "TheMsg");
comm->Publish("TheMsg", 12, (unsigned char*) "Hello world");
```

This will set up a connection between this module and the module "friend." It will then subscribe "friend" to this modules "TheMsg" and publish the information to "friend" rather than send it. This allows another, anonymous module, to externally subscribe to "TheMsg" and thus wiretap messages of type "TheMsg" between this module and "friend." This mechanism comes, of course, at the expense of some additional message overhead on the part of the sender.

Using multiple IPT servers

One way to implement multiple communications "domains" is to have one central server, and then append domain names to module names to give each module a system wide unique name. For example, if you wanted to connect to the module "Navigator" in the domain "VehicleA", you would just concatenate the two names together to request a connection to "VehicleA:Navigator".

The problem with this "one server, many domains" approach is that there is usually a good physical reason for splitting a system into multiple domains. One sample system has two mobile robots and a fixed base connected together by relatively slow radio links. Now, if we run one IPT server on the central platform for the entire system we run into huge problems. Every module on the vehicles has to go across a slow radio link in order to register messages and request connections. By having one IPT server for each platform, with each platform being a "domain" we can drastically reduce the overhead involved in starting up a system.

In a multi-domain system each domain will have an IPT server, and each IPT server is given a list of "peer" domains (and the machines on which these peer domain servers are running). A domain server hooks up to other domain servers as it becomes necessary. Domain servers interact with each other to direct interdomain connection requests and to make sure that messages are referred to consistently across domains.

In a multi-domain system each server must have a domain "name" associated with it. This name is passed in using the **-d** flag,

```
  iptserver -d domain_name.
```

Modules in one domain can connect to a module in another domain by appending the domain name and a colon to the desired module name.

For example, say a module named "Display" in domain "VehicleA" desires to connect to the module named "Display" in the domain named "VehicleB." To do this, VehicleA's Display would have to execute something like,

```
  IPConnection* veh_b_display = comm->Connect("VehicleB:Display");
```

This request would go to the domain server for VehicleA, which would in turn make the request of the domain server for VehicleB. The result will be a direct connection between VehicleA's Display and VehicleB's Display.

The **-C** flag is used to specify the domains that a server can initiate connections to.

```
iptserver -d domain_name -C peer_name machine_name ...
```

The **-C** flag must be followed by the peer domain name and then the machine on which that domain server is running. You can give any number of **-C** flags. If you use the **-C** flag to specify peer names you must use the **-d** flag to specify this domain's name.

Integrating IPT with GUI's

A good way to build a user interface is as a separate process communicating with the process that do the work via IPT. This approach means you will have to integrate IPT with whatever graphical user interface (GUI) system you have. Here is an example of the most straightforward, and least appealing, way to do this melding:

```
while (1) {
     handle_GUI_events();
     comm->Idle(SLEEP_TIME);
}
```

This example handles whatever events the GUI system produces, and then waits in an IPT idle loop for a certain amount of time. The problem is, you are trading off needlessly burdening the CPU with jerky response. As **SLEEP_TIME** is reduced, the response to the graphics events becomes smoother, but the amount of CPU time spent cycling increases.

IPT provides two better ways to integrate itself with GUI systems, and both of them take advantage of the fact that many GUI systems, such as those built on **X**, have at their roots sockets and file descriptors, just like the IPT connections on a UNIX machine. IPT either lets you add an "administrative" connection to its internal connection list, or you can get access to every connection that IPT administers, and thus you can add the file descriptors of those connections to the GUI's internal file descriptor list. Then, instead of switching back and forth between two procedures that check for input and incoming events, you only have to sit in one "event loop," which will monitor for inputs from all sources without any busy waiting.

First, how to add arbitrary file descriptors to the list of file descriptors that IPT maintains.

C++ Function
AddAdministrative Connection
(ipt/ipt.h)

```
void IPCommunicator::AddAdministrativeConnection(int fd,
                                     IPConnectionCallback* activity_cb,
                                     IPConnectionCallback* closing_cb)
```

This method adds an administrative connection with file descriptor *fd* to the list maintained by IPT. When activity is detected on this connection, the **Execute** method of the *activity_cb* callback will be invoked if *activity_cb* is not NULL. If IPT detects that the file descriptor has been closed by an external source or it wants to close it, then the **Execute** method of the *closing_cb* callback will be invoked if *closing_cb* is not NULL.

C Function
iptAddAdministrative Connection
(ipt/ipt.h)

```
void iptAddAministrativeConnection(int fd,
                          void (*activity_func)(IPCommunicator*,
                                                IPConnection* conn,
                                                void* data)
                          void (*closing_func)(IPCommunicator*,
                                                IPConnection* conn,
                                                void* data),
                          void* data)
```

This is the C cover function for **IPCommunicator::AddAdministrativeConnection**. It creates callbacks from *activity_func* and *closing_func*, if they are non-NULL. When *fd* has activity, *activity_func* is called with data being equal to the data you passed in. Similarly, when *fd* is shutdown and *closing_func* is non-NULL, *closing_func* is called with *data* set to the *data* you passed in.

C++ Function
RemoveAdministrative Connection
(ipt/ipt.h)

```
void IPCommunicator::RemoveAdministrativeConnection(int fd)
```

This method removes the administrative connection with file descriptor *fd*. The closing callback is not invoked.

C Function
iptRemoveAdministrative Connection
(ipt/ipt.h)

```
void iptRemoveAdministrativeConnection(int fd)
```

This is the C cover function for **IPCommunicator::RemoveAdministrativeConnection**.

Example

You can use these routines to add a file descriptor associated with a GUI to be monitored by IPT. In the following example, we assume the GUI is encapsulated by a class which has methods for returning its file descriptor and for handling events.

```
// class to handle activity on a GUI connection by invoking the GUI's
// event handling method
class HandleGUIConnection : public IPConnectionCallback {
  public:
    HandleGUIConnection(GUI* gui) { _gui = gui; }
    virtual void Execute(IPConnection*) { gui->HandleEvents(); }
  private:
    GUI* _gui;
};

void RegisterGUI(IPCommunicator* comm, GUI* gui)
{
     // add an administrative connection to handle GUI input
     IPConnectionCallback* cb = new HandleGUIConnection(gui);
     comm->AddAdministrativeConnection(gui->FileDescriptor(),
                                        cb, NULL)
}
```

It should be noted that this approach will not work perfectly if the GUI is not completely socket based. For example, this is the case for SGI systems, because there are events that need to be handled that are not accompanied by activity on the graphics file descriptor.

In situations like that, you will need to use the second method for integrating with a GUI, which is to feed all of IPT's file descriptors to the GUI.

C++ Function
IterateConnections
(ipt/ipt.h)

```
void IPCommunicator::IterateConnections(IPConnectionCallback* cb)
```

This method will invoke *cb*'s **Execute** method for every connection that IPT manages.

C Function
iptIterateConnections
(ipt/ipt.h)

```
void iptIterateConnections(IPCommunicator* comm,
                           void (*func)(IPCommuncator* comm,
                                        IPConnection* conn,
                                        void* data),
                           void* data)
```

This is the C cover function for **IPCommunicator::IterateConnections**.

Example.

The **IterateConnections** method can be used to add all of the file descriptors managed by IPT to a GUI. You will also need to use the module connection and disconnection callbacks to register modules as they connect and disconnect. We will again use the abstract "GUI" class, with additional methods for adding and removing file descriptors (with callback functions). Most socket based GUI systems have a method for adding file descriptors to be managed with callbacks.

```
// This routine will be called whenever the GUI detects activity on
// any IPT file descriptor. The Idle method will process all of the
// input coming into IPT
void gui_reports_activity(int fd, void* callback_data)
{
     IPCommunicator* comm = (IPCommunicator*) callback_data;
     comm->Idle(0.0);
}

class GUIModuleConnecting : public IPConnectionCallback {
  public:
     GUIModuleConnecting(GUI* gui) { _gui = gui; }
     virtual void Execute(IPConnection* conn)
          { _gui->AddFileDescriptor(conn->FD(),
                                      (void*) conn->Communicator()); }
  private:
     GUI* _gui;
};

class GUIModuleDisconnecting : public IPConnectionCallback {
  public:
     GUIModuleDisconnecting(GUI* gui) { _gui = gui; }
     virtual void Execute(IPConnection* conn)
          { _gui->RemoveFileDescriptor(conn->FD()); }
  private:
     GUI* _gui;
};
void RegisterWithGUI(IPCommunicator* comm, GUI* gui)
{
     IPConnectionCallback* reg_cb = new GUIModuleConnecting(gui);

     comm->IterateConnections(reg_cb);
     comm->AddConnectCallback(reg_cb);
     comm->AddDisconnectCallback(new
GUIModuleDisconnecting(gui));
}
```

In this example, whenever input appears at any of IPT's managed file descriptors, that input will trigger the GUI to invoke the routine **gui_reports_activity**, which in turn will use the **IPCommunicator::Idle** method to process any incoming IPT events.

This method will only work if all connections managed by IPT are based on file descriptors. If this is not true, as is the case of subclasses of **IPCommunicator** that use shared memory connections, then this method will not work, because those non-file descriptor based connections will always report file descriptors of -1. This is not too large a concern, since most non-file descriptor based connections will only be implemented on real time operating systems such as VxWorks, and modules that run on these real time operating systems would not be wise places to implement graphical user interfaces anyway.